

Chapter 3

Nonrelational Databases in Azure

MICROSOFT EXAM OBJECTIVES COVERED IN THIS CHAPTER:

- ✓ Describe nonrelational data workloads.
 - Describe the characteristics and types of NoSQL data.
 - Recommend the correct NoSQL database.
 - Determine when to use a NoSQL database.
- ✓ Describe nonrelational data offerings on Azure.
 - Identify Azure data services for NoSQL workloads.
 - Describe Azure Cosmos DB APIs.
 - Describe Azure Table storage.
- ✓ Identify basic management tasks for nonrelational data.
 - Describe provisioning and deployment of NoSQL data services.
 - Describe method for deployment including the Azure portal, Azure Resource Manager templates, Azure PowerShell, and the Azure command-line interface (CLI).
 - Identify data security components (e.g., firewall, authentication, encryption).
 - Identify basic connectivity issues (e.g., accessing from on-premises, access with Azure VNets, access from Internet, authentication, firewalls).
 - Identify management tools for NoSQL databases.





Nonrelational databases, also commonly referred to as NoSQL databases, allow users to **store and query** nonrelational data **without needing** to mold it to fit a predefined schema. NoSQL databases are typically used in scenarios where data needs to be **ingested and read very quickly**, such as gaming, e-commerce, IoT, and mobile applications. The main categories of NoSQL databases are key-value, document, columnar, and graph. This chapter will discuss the different categories of NoSQL databases, how they can be implemented using Azure Cosmos DB, and basic management tasks for NoSQL databases in Azure.

Nonrelational Database Features

With the boom of data-driven applications over the last several years, organizations have had to **reconsider** how they store data. Large volumes of data coming in all shapes and sizes needing to be captured in near real time **make it nearly impossible** for organizations to use traditional relational models for all their data storage needs. Additionally, the advent of cloud computing enabled organizations to **easily**, and **cheaply**, scale their data storage solutions **horizontally** across different geographic regions. This allows organizations to store data in its **natural format** without needing to apply complex data normalization rules first. For these reasons, NoSQL databases have become a **popular choice** for software developers who require a dynamic data storage solution.

Instead of forcing data to fit a rigid schema, NoSQL databases use a storage model that is optimized for the requirements of the **data being stored**. Not needing to focus so much on database management empowers software developers to build applications with a **more agile** approach, allowing them to **adapt to changing requirements** more quickly.

While there are several categories of NoSQL databases, they share the following characteristics:

- **Ambiguous** implementation of ACID principles. This is a benefit for **transactional workloads** where there are high volumes of data being processed at very fast speeds.
- **Easily scaled horizontally** across multiple partitions and storage devices since there are no relationships between data, allowing data to reside anywhere.
- **Schema flexibility** that enables **faster and more agile** software development. This allows new data records to have different fields and data types than previously stored records. The flexible schema design inherent to NoSQL databases makes them ideal for **semi-structured and unstructured data**.

Generally, NoSQL databases can be categorized as either key-value stores, document databases, columnar databases, or graph databases. These were summarized in Chapter 1 and are detailed in the following sections.

Key-Value Store

Key-value stores are the **simplest type** of NoSQL database and store pieces of data as two common elements: a **unique key** for identification and the **value** that is captured. Keys can be used by applications to **perform lookup operations** to retrieve the data values that are associated with them. These data stores are **highly scalable**, distributing data across all available storage by applying a **hash algorithm** to the keys. While keys are **unique and scalar**, values can range from scalar values to **complex data** objects such as JSON arrays. Figure 3.1 is an example of a key-value store that stores phone directory data.

FIGURE 3.1 Key-Value store

Key	Value
Pete	{{(012) 123-4567}}
Jim	{{(987) 765-4321}}
Kate	{{(654) 879-1234, (123) 456-7890}}

Key-value stores are optimized for ingesting **large volumes** of data that must be stored and read **very quickly**. Applications reading data from key-value stores typically perform **simple lookups** using a single key or a range of keys. Here are two common scenarios where key-value stores are ideal storage solutions:

- **Web applications** that store user session metadata in real time. These applications can also use key-value stores to make **real-time recommendations** to users as they are browsing the site.
- **Caching frequently accessed data** to optimize application performance by minimizing reads to disk-based storage such as Azure SQL Database.

While key-value stores are great at serving data to applications performing **simple read operations**, they are **not ideal** storage solutions for applications that need to perform **intense search operations**. They also do not support scenarios where queries need to filter data by the values. Key-value stores also **only support insert and delete operations**, requiring users to modify data by completely overwriting existing items.

Azure provides a few different options for implementing a key-value store:

- Azure Table storage
- Azure Cosmos DB Table API
- Azure Cache for Redis

This chapter will focus on Azure Table storage and the Azure Cosmos DB Table API as these are in scope for the DP-900 exam. You can find more information at <https://docs.microsoft.com/en-us/azure/azure-cache-for-redis/cache-overview> if you would like to learn more about Azure Cache for Redis.

Document Database

Document databases are sophisticated versions of key-value stores that store data as semi-structured documents. Individual data fields that are contained in a document can consist of singleton values or a nested group of elements. Documents are assigned unique IDs that can be used by applications to query data. The unique ID, also known as a document key, is often hashed to distribute data evenly across available storage. Figure 3.2 illustrates an example of how a document database is structured.

FIGURE 3.2 Document database

Key	Document
1001	<pre>{ "CustomerID": 101, "OrderItems":[{ "ProductID": 500, "Quantity": 2, "Cost": 350 }, { "ProductID": 505, "Quantity": 1, "Cost": 50 }], "OrderDate": "2021-07-14" }</pre>
1002	<pre>{ "CustomerID": 102, "OrderItems":[{ "ProductID": 450, "Quantity": 5, "Cost": 650 }], "OrderDate": "2021-07-15" }</pre>



While most document databases use JSON format, there are some that encode data in other formats, such as XML, YAML, BSON, or plain text.

Documents contain all the data for a given entity. One example of an entity could be all details related to an order made on an e-commerce site such as the customer’s information and the items ordered. This denormalized way of storing data can optimize the performance of queries that need to retrieve data very quickly by eliminating the need to read data from multiple tables.

Each document in a document database can have **different sets of fields**, like values in a key-value store. Unlike key-value stores, however, queries can **filter** data by **field values**. This makes document databases ideal candidates for applications that must **store large amounts** of data very quickly and need to perform **sophisticated filters** when querying the data.

Azure provides a couple of different options for implementing a document database:

- Azure Cosmos DB Core (SQL) API
- Azure Cosmos API for MongoDB

Columnar Database

Columnar databases are like relational databases in that they organize data into **columns and rows**. Unlike relational databases, columnar databases are completely **denormalized**, dividing data into groups known as **column families**. Column families contain data that would normally be separated into a set of columns if the data was stored in a relational database. However, unlike in a relational database, rows in a column family **do not have to share** a common schema. One entry can have several columns, while another might only have one or two.

Column families that are a part of the same entity share a common row key. This key is considered the primary key and is used to physically store data in order. Applications can **perform lookups** using a specific **row key** or a **range of keys**. Secondary indexes can also be applied to allow applications to filter by column values.

Figure 3.3 illustrates an example of a columnar database used by a company that sells bicycles and bicycle accessories. This database has two column families, one with product information and another listing quantity information. Related column families are bound by a common product key.

FIGURE 3.3 Columnar database

Row Key	Column Families	
ProductKey	ProductInfo	Quantity Info
500	Category: Bicycle Subcategory: Mountain Bike Color: Matte Black UnitPrice: 700	QuantityOnHand: 10 QuantitySold: 12 ProductRating: 8.2
505	Category: Helmet Subcategory: Standard Helmet Color: Orange UnitPrice: 30	QuantityOnHand: 30 QuantitySold: 40 ProductRating: 9.3

Columnar databases are typically used in **analytics** scenarios. Grouping data into column families allows queries to **jump directly** to where specific pieces of data are located. This can result in very **fast aggregations** since the query does not need to jump from row to row to find the field that is a part of the aggregation. Columns in a column family are also of the same data type, resulting in better **data compression** and **faster queries**.

While columnar databases are optimal for analytical workloads that aggregate data, they are not well-suited for transactional workloads where queries perform value-specific lookups. This is where a traditional relational database that stores data in a row-wise format is more performant. Writing data to a columnar database can also take more time than in a row-wise database. While new entries in a row-wise database can be inserted in one operation, columnar databases write new entries to each column one by one.

Azure provides a couple of different options for implementing a columnar database:

- Azure Cosmos DB Cassandra API
- HBase in Azure HDInsight

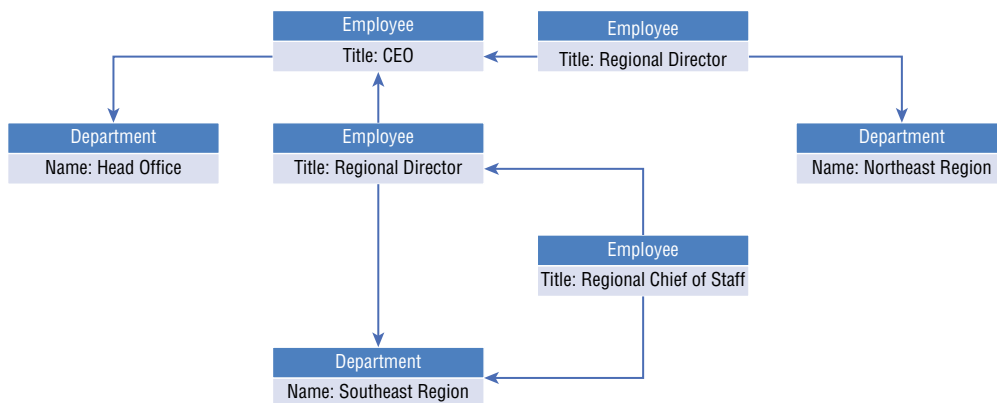
This chapter will focus on the Azure Cosmos DB Cassandra API. You can find more information at <https://docs.microsoft.com/en-us/azure/hdinsight/hbase/apache-hbase-overview> if you would like to learn more about HBase in Azure HDInsight.

Graph Database

Graph databases are specialized databases that focus on storing the relationships between data entities. Entities in a graph database are stored as nodes, while the relationships between entities are referred to as edges. Nodes and edges can contain attributes specific to them, like tables in a relational database. Edges can also have a direction to indicate the nature of a relationship between two nodes.

Applications that use graph databases run queries that need to traverse the network of nodes and edges, analyzing the relationships between entities. Figure 3.4 demonstrates an example of a graph database that stores information about an organization's personnel chart. The entities represent different job titles and departments, while the edges represent the reporting structure for different employees.

FIGURE 3.4 Graph database



Graph databases are optimal for solutions that ask questions such as “Find all employees that report directly to the CEO.” Applications querying large graphs with lots of nodes and edges, such as social media networks, can perform complex analyses very quickly. While relational databases can be used to store the same data as a graph database, queries written for the graph database circumvent any join operations or subqueries that would need to be considered for the relational database.

Graph databases can be implemented in Azure using the Azure Cosmos DB Gremlin API.

Azure Cosmos DB

Azure Cosmos DB is a multi-model PaaS NoSQL database management system. *Multi-model* means that organizations can use Azure Cosmos DB to build key-value, document, columnar, and graph data stores. The different categories are made available as database APIs, including the Table API, Core (SQL) API, API for MongoDB, Cassandra API, and Gremlin API. Users will have the option of choosing one of these APIs when deploying an instance of Azure Cosmos DB.

The highest level of management for Azure Cosmos DB is a database account. Currently, you are allowed to have up to 50 Azure Cosmos DB accounts in an Azure subscription, but that can be increased by submitting a support ticket in the Azure Portal. Each database account can have one or more databases (referred to as a keyspace when using the Azure Cosmos DB Cassandra API), that serve as the unit of management for a set of containers.

Containers are the fundamental unit of scalability for throughput and storage. It is at this level that data is partitioned and replicated across multiple regions. Users can also register stored procedures, user-defined functions, triggers, and merge procedures within a container. Containers are identified by different names depending on which type of NoSQL database is deployed. Table 3.1 lists the naming convention used by each Azure Cosmos DB API.

TABLE 3.1 Azure Cosmos DB API-specific names for containers

API	Container Naming Convention
Table API	Table
Core (SQL) API	Container
Cassandra API	Table
API for MongoDB	Collection
Gremlin API	Graph

Data stored in containers is automatically grouped into logical partitions based on a partition key and is distributed across physical partitions. A *partition key* is a designated data field that is used to efficiently group throughput and related data. Other than choosing an appropriate partition key, partition administration is handled internally by Azure Cosmos DB.

Individual data records stored in a database container are referred to as *items*. When Azure Cosmos DB partitions data, it groups items with the same partition key value into the same logical partition. Items are automatically indexed as they are added to a container. Indexing behavior can also be customized by configuring the indexing policy on the container. Like containers, items are referred to by different names depending on which type of NoSQL database is deployed. Table 3.2 lists the naming convention used by each Azure Cosmos DB API.

TABLE 3.2 Azure Cosmos DB API-specific names for items

API	Item Naming Convention
Table API	Entity
Core (SQL) API	Item
Cassandra API	Row
API for MongoDB	Document
Gremlin API	Node or edge



When choosing a partition key for your container, be sure to select a field that has a high range of values. This will ensure that data is evenly distributed among partitions. Partition key values also cannot be updated, so be sure to select a column that has values that do not change.

Azure Cosmos DB ensures that data is highly available, regardless of which API is being used. As a matter of fact, Azure Cosmos DB guarantees 99.99 percent high availability when deployed to a single region and 99.999 percent high availability when deployed to multiple regions. Reads and writes are also guaranteed within 10 milliseconds across regions wherever the data is being replicated to. The following sections focus on how to optimize performance and availability for Azure Cosmos DB before finishing with an overview of the different database APIs.

High Availability

High availability is a foundational component of Azure Cosmos DB. Global distribution in Azure Cosmos DB allows users to easily replicate data to multiple regions by associating

one or more additional regions to an Azure Cosmos DB account. Adding new regions can be done through the **Azure Portal** or **programmatically**.

New regions can be configured to be **read-only** or to allow **both reads and writes**. Read-only workloads such as those produced from reporting applications can be offloaded to the read-only replicas, resulting in better performance for these workloads and those performing write operations. It is recommended to configure **at least two different regions** to allow writes in case of regional failure. This guarantees that if the primary region goes down, then write operations will **automatically** be **routed** to another region.

In addition to global distribution, Azure Cosmos DB maintains **four replicas** of the data within **each region**. For example, if you define an Azure Cosmos DB account to use two regions, then eight copies of the data will be maintained. **Data resiliency** within regions can also be guaranteed by enabling **availability zone support**. Availability zones ensure that **replicas** are placed in **different** zones of a given **region**, protecting data from **in-region zonal failures**.

Consistency Levels

Distributed databases such as Azure Cosmos DB that manage multiple write copies across different regions requires a **trade-off** between data consistency, availability, and performance. Using a **strong consistency model** results in the most updated data being read by applications but can result in **slower performance** since data has to replicate and be committed to each associated region before an application is allowed to read data. While eventual consistency offers better performance, applications reading data are at risk of returning **dirty data**.

Azure Cosmos DB offers **five** well-defined *consistency levels* to balance the trade-off between consistency, availability, and performance. The following list describes each consistency level, starting with the strictest level of consistency and finishing with the most relaxed:

- **Strong**—This consistency level guarantees that reads return the **most recent version of data**, regardless of what region an application is reading data from. There is a possibility of **slower performance** as application connections may experience delays until transactions are committed across every associated region.
- **Bounded staleness**—With this consistency level, applications reading data from the **same region** that it was written to use **strong consistency**. For applications reading data from regions outside of where data was written, there is a set “**staleness window**” for data. This means that data is **committed asynchronously** to other associated regions in the Azure Cosmos DB account. The staleness window can be configured one of two ways:
 - The number of versions of a record
 - A set time interval between writesWhenever either of these two **limits** is **reached**, **connections are paused**, and **data is committed** to the outside regions. Bounded staleness is a good choice for applications that require **low write latency** and **guaranteed local consistency**.
- **Session**—This is the most widely used consistency level for single region and globally distributed accounts. It grants a **session token** to the application writing data and

guarantees that it and any other application sharing the same session token see the most recent version of data. All other reads use eventual consistency.

- *Consistent prefix*—Data is eventually replicated across regions but this level does not provide any guarantees on how long it will take for data to be replicated. However, it does guarantee that applications reading data will read data in order. For example, if an application writes records 100, 101, and 102 to a database in that order, then an application reading the data will see either 100, [100, 101], [101, 102] or [100, 101, 102], but never out-of-order combinations like [100, 102].
- *Eventual*—Much like the consistent prefix consistency level, eventual consistency guarantees that data will eventually be replicated across regions but does not provide any guarantees on how long it will take for data to be replicated. There is also no guarantee that data will not be replicated out of order. While it is the most performant consistency level in terms of read and write latency, it is the least likely to guarantee consistency.

Session consistency is the default consistency level for Azure Cosmos DB. You can change the consistency to be more consistent or more performant with a sliding scale in the Azure Portal or with a deployment script.

Request Units

As with any PaaS offering in Azure, Azure Cosmos DB abstracts compute resources from the end users. Instead, resources such as CPU, IOPS, and memory are bundled into units of measure called Request Units.

Request Units (RUs) represent the throughput required to read and write data in Azure Cosmos DB. One general rule of thumb is that the cost to read and write a 1 KB item is approximately 1 RU and 5 RUs respectively. This is true regardless of what type of Azure Cosmos DB API you are interacting with. Of course, the number of RUs that are required for a given query are going to vary based on the volume of data read or written, how well the data is indexed, the consistency model choice, and the complexity of the query. More information on RU considerations can be found at <https://docs.microsoft.com/en-us/azure/cosmos-db/request-units#request-unit-considerations>.



The number of RUs used by a query will vary depending on how evenly distributed data is across partitions. Skewed data caused by a poorly chosen partition key can cause queries to have to perform cross-partition searches, which can take significantly more time and throughput than a query interacting with a single partition.

Since RU usage is measured per second, throughput is set using the Request Units per second (RU/s) measurement. The way throughput is charged depends on the way the RU/s measurement is configured on the Azure Cosmos DB account. These include provisioned throughput, autoscale, and serverless.

Provisioned

Azure Cosmos DB allows users to manually increase or decrease the number of RU/s. With this option, users can allocate RU/s at the database level and at the container level through the Azure Portal or programmatically using the .NET or Java SDK. The minimum throughput that can be allocated for a container or database is 400 RU/s.

Provisioned throughput on an Azure Cosmos DB database is shared across all the containers in the database. This means that all containers share the compute resources that are allocated to a database. There is also an option to dedicate throughput to specific containers. For example, let's say that you create a database with five containers and one of these containers requires dedicated throughput. When provisioning this container, you can enable the *Provision dedicated throughput for your container* option to explicitly allocate RU/s to the container. The rest of the containers will share the throughput allocated to the database.

There are a couple of caveats regarding the 400 RU/s minimum for provisioned throughput. The actual minimum is typically the maximum of the following:

- 400 RU/s
- 10 RU/s per every 1 GB added to storage
- The highest number of RU/s previously provisioned divided by 100

For example, if 50 GB of data is added to a new container, then the minimum RU/s for that container is 500 RU/s.

Provisioned throughput is difficult to calculate when first deploying a database. It's important to understand how much data you will be storing, how many containers you will need, and what type of queries will be interacting with the database. A helpful tool for estimating throughput and throughput cost can be found at <https://cosmos.azure.com/capacitycalculator>. The calculator uses parameters such as the number of regions, whether there are additional write regions, the volume of data stored, and the number of create, read, and delete operations per second to estimate the number of RU/s needed and how much the workload will cost.

Autoscale

Autoscale is a version of provisioned throughput that grants Azure Cosmos DB the ability to automatically scale the throughput of a database or container. Once it's enabled, users can set the maximum number of RU/s that a database or container can scale to. Throughput is then scaled based on usage without impacting the performance of any existing workloads.

Typical use cases for autoscale include workloads with inconsistent or infrequent usage and new applications where the user is not sure how much throughput to provision. Autoscale is a simple, cost-effective solution for most workloads, while still providing high availability.

Serverless

Serverless mode does not require users to provision throughput when creating databases or containers in Azure Cosmos DB. Instead, the Azure Cosmos DB manages throughput for

workloads and bills users at the end of their billing period for the number of RU/s that were consumed during that period.

Scenarios that are best suited for serverless include those where you expect unpredictable workload traffic with **long periods of downtime**. These include development, test, and prototyping scenarios with **unknown traffic patterns** and applications that have **highly random activity**.



Provisioned throughput, autoscale, and serverless are available for all five Azure Cosmos DB APIs.

Azure Cosmos DB APIs

Azure Cosmos DB offers multiple database APIs to create different types of NoSQL databases, including the following options:

- Table API for key-value stores
- Core (SQL) API for document databases
- API for MongoDB for document databases
- Cassandra API for columnar databases
- Gremlin API for graph databases

Users are asked to select an API when creating an Azure Cosmos DB account for the first time. Choosing the most appropriate API depends entirely on the solution(s) that instance of Azure Cosmos DB will be supporting. The following sections will discuss each API and when to use them.


Table API

The Azure Cosmos DB Table API is a **key-value store** that is based on **Azure Table storage**. The differences are primarily focused on features that are inherent to the Azure Cosmos DB service such as **higher performance and availability**, **global distribution**, **automatic secondary indexes**, and **more options for configuring throughput**. However, it is important to know the core components of Azure Table storage to understand how to implement a key-value store with the Azure Cosmos DB Table API.

Azure Table storage is a **key-value store** that **stores nonrelational, structured data**. Containers in Azure Table storage are represented as **tables** and can be created in an Azure storage account. Data is stored in tables as a **collection of entities**, like rows in a relational database. Individual data fields in entities are represented as **properties**. Properties are like **columns** in a relational database.

While the terminology may present Azure Table storage as a relational data store, it is far from it. Tables **do not enforce** a schema on entities, allowing entities to have different sets of properties. However, there are some entity conditions that must be adhered to. First, each entity must include a set of system properties that specify a **partition key**, a **row key**, and a

time stamp. Second, there is also a 255-property limit that includes the three previously mentioned system properties.



Partition keys uniquely identify each partition in a table. Row keys uniquely identify each entity in a partition. Together, the two keys form primary keys that uniquely identify each entity in a table.

Typical use cases for Azure Table storage include caching user data for web applications, address books, device information, or other types of metadata. Applications written in .NET, Java, Python, Node.js, or Go can interact with Azure Table storage using the Azure SDK for those languages. Data can be accessed using OData for all languages and LINQ queries for applications written in .NET.

The Table storage service in Azure is moving from Azure Storage to Azure Cosmos DB to overcome existing limitations with latency, scaling, throughput, availability, and query performance. Keep in mind that Azure Cosmos DB Table API and Azure Table storage share the same data model as Azure Table storage and expose the same query operations through their SDKs. For this reason, applications written for Azure Table storage can easily migrate to the Azure Cosmos DB Table API with minimum code changes.

Table 3.3 includes a list of some of the primary benefits that can be gained by migrating to the Azure Cosmos DB Table API from Azure Table storage.

TABLE 3.3 Azure Table storage vs. Azure Cosmos DB Table API

Feature	Azure Table Storage	Azure Cosmos DB Table API
Maximum Entity Size	1 MB	2 MB
Latency	Fast, but no upper bounds on latency.	Less than 10 ms latency for reads and writes at the 99th percentile, anywhere in the world.
Throughput	Variable throughput model.	Highly scalable with provisioned, autoscale, and serverless throughput options.
Global Distribution	Single region and one optional secondary read region.	Support for multi-region writes and reads with automatic and manual failovers at any-time, anywhere in the world.
Consistency	Strong within the primary region and eventual in the secondary.	Five well-defined consistency levels.

More benefits can be found at <https://docs.microsoft.com/en-us/azure/cosmos-db/table/introduction#table-offerings>.

Core (SQL) API

The Azure Cosmos DB Core (SQL) API is a **document database service** and is the **default** API for Azure Cosmos DB. As the name implies, the Core (SQL) API is the core, or **native**, API for working with NoSQL data in Azure Cosmos DB. This API is recommended for **new applications** that require **high performance and global distribution** and when **migrating to Azure from other NoSQL database platforms**. The Core (SQL) API is also the recommended **migration option** for relational databases that require the **benefits of a NoSQL database**.

The data model for the Core (SQL) API uses the **default hierarchy** of resources where an account hosts one or more databases, a database hosts a set of containers, and a container stores data as items. Items are formatted as **JSON** documents and can be interacted with using **SQL**.

SQL syntax used by the Core (SQL) API is very **familiar to T-SQL** with some additional functionality that is specialized for interacting with JSON data. For example, an application that is querying the Persons container of a database in an Azure Cosmos DB Core (SQL) API account might want to retrieve information about the user stored in the following JSON document:

```
{
  "firstname": "John",
  "lastname": "Smith",
  "age": 23,
  "favoriteSports": {
    "mostFavorite": "Basketball",
    "secondFavorite": "Baseball"
  },
  "id": "de5760d6-64fd-4dc3-8cb9-cc914ee860b0",
}
```

The application can use the following SQL query to return the user's first name and their most favorite sport:

```
SELECT p.firstname, p.favoriteSports.mostFavorite
FROM Persons AS p
WHERE p.id = 'de5760d6-64fd-4dc3-8cb9-cc914ee860b0'
```

The results from the query are:

```
[
  {
    "firstname": "John",
    "mostFavorite": "Basketball"
  }
]
```

Along with SQL, the Core (SQL) API supports **user-defined functions** and **stored procedures** written in JavaScript.

API for MongoDB

MongoDB is a popular **document database platform** that stores data items as **Binary JSON** (BSON) documents. Organizations wanting to take advantage of the **scalability, performance, high availability, and ease of maintenance** that Azure Cosmos DB provides without changing any existing code can do so by migrating their MongoDB databases to the Azure Cosmos DB API for MongoDB.

Data can be migrated to the API for MongoDB using tools such as **mongodump**, **mongorestore**, or the Azure native **Azure Database Migration Service**. Once the data is in Azure, organizations can continue using their **existing MongoDB applications** by just changing the **connection string**. Tools that are native to MongoDB such as the **MongoDB shell** and **MongoDB Compass** can interact with databases hosted on the API for MongoDB just as they would with MongoDB hosted in an on-premises datacenter.



MongoDB uses a proprietary query language to perform read and write operations. Syntax and examples of queries written in the MongoDB Query Language can be found at <https://docs.mongodb.com/manual/crud>.

The Azure Cosmos DB API for MongoDB is compatible with MongoDB server versions 4.0, 3.6, and 3.2. More information about migrating to the Azure Cosmos DB API for MongoDB and estimating throughput can be found at <https://docs.microsoft.com/en-us/azure/cosmos-db/mongodb/mongodb-introduction>.

Cassandra API

Apache Cassandra is a popular **columnar database** that stores large volumes of data using a **column-oriented schema**. Just as with MongoDB and the Azure Cosmos DB API for MongoDB, organizations can migrate their existing Cassandra workloads to the Azure Cosmos DB Cassandra API to take advantage of the **premium capabilities** that Azure Cosmos DB provides.

Users can query data stored in the Cassandra API using the **Cassandra Query Language** (CQL) and tools like the **CQL shell** (cqlsh). Applications can also continue to use existing Cassandra client drivers to interact with Cassandra databases hosted on the Cassandra API.

Gremlin API

The Azure Cosmos DB Gremlin API uses the **Apache Tinkerpop graph** framework to provide a **graph database interface** in Azure Cosmos DB. It allows organizations to manage existing and new graph database applications without needing to worry about overhead such as **infrastructure, throughput, and availability**.

While data is stored as **JSON documents** as they are with the Core (SQL) API, the Gremlin API enables the data to be queried with **graph queries**. Applications can query databases hosted on the Gremlin API using the **Gremlin query language**. More information on the Gremlin query language and using it to query data stored in the Azure Cosmos DB Gremlin API can be found at <https://docs.microsoft.com/en-us/azure/cosmos-db/graph/tutorial-query-graph>.

Management Tasks for Azure Cosmos DB

Just as with relational PaaS databases in Azure, there are several management tasks that must be taken into consideration for Azure Cosmos DB. These include deploying instances of Azure Cosmos DB, configuring throughput and global distribution, migrating existing on-premises workloads to Azure Cosmos DB, and maintaining data security. The following sections will discuss these tasks in detail, as well as how to troubleshoot common connectivity issues when using Azure Cosmos DB.

Deployment Options

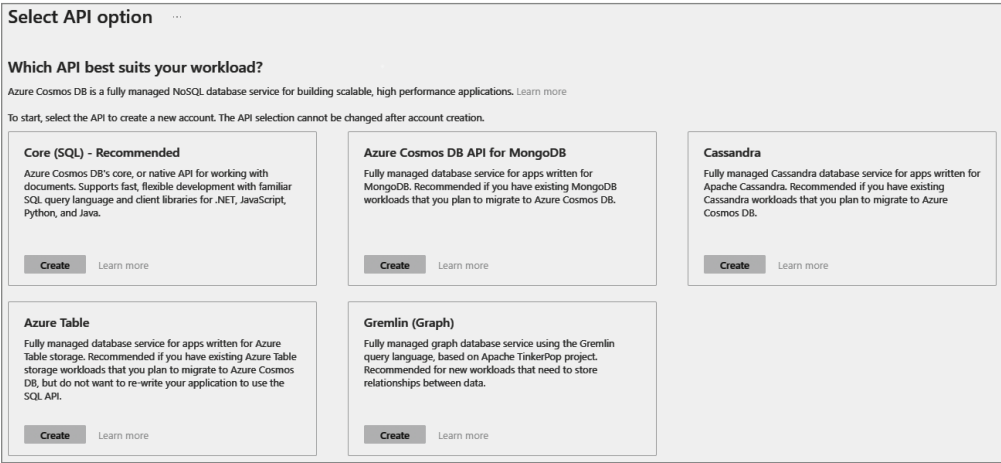
Instances of Azure Cosmos DB can be deployed to an Azure subscription using several methods. Users can manually configure the necessary requirements for their Azure Cosmos DB environment through the Azure Portal or automate the deployment with different scripting languages. As discussed in Chapter 2, Azure PowerShell, Azure CLI, and ARM templates are some of the most common ways to automate Azure resource deployments. Let's discuss in the following sections how to use these different options to configure and deploy Azure Cosmos DB.

Azure Portal

Use the following steps to create an Azure Cosmos DB account through the Azure Portal:

1. Log into `portal.azure.com` and search for *Azure Cosmos DB* in the search bar at the top of the page. Click *Azure Cosmos DB* to go to the Azure Cosmos DB page in the Azure Portal.
2. Click **Create** to start choosing the **configuration options** for your Azure Cosmos DB account.
3. The first requirement for creating an Azure Cosmos DB account is to select the most appropriate **API** for the workload it will be serving. The **Select API Option** page allows you to choose from one of the five APIs. Figure 3.5 is a screen shot of what this page looks like. For the purposes of this example, we will select the *Core (SQL)* API.
4. The Create Azure Cosmos DB Account page includes six tabs with different configuration options to tailor the Azure Cosmos DB account to fit your needs. Let's start by exploring the options available in the Basics tab. Along with the following list that describes each option, you can view a completed example of this tab in Figure 3.6.
 - a. Choose the **subscription and resource group** that will contain the Azure Cosmos DB account. You can **create a new resource group** on this page if you have not already created one.
 - b. Enter a name for the Azure Cosmos DB account.

FIGURE 3.5 Select Azure Cosmos DB API.



- c. Choose the **primary Azure region** for the account.
- d. Choose whether you want to **provision throughput** for Azure Cosmos DB or have Azure Cosmos DB manage throughput with serverless.
- e. The last option allows you to choose whether you would like to apply the **free tier** discount to this Azure Cosmos DB account. This allows you to get the first 1000 RU/s and 25 GB of storage for free in the account. This option can be enabled for one account per subscription.

FIGURE 3.6 Create an Azure Cosmos DB Account: Basics tab.



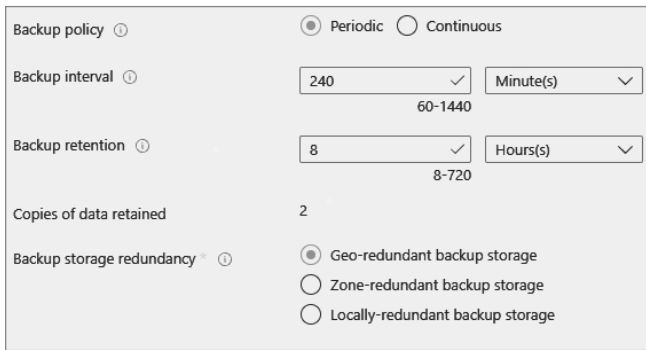
- 5. The Global Distribution tab allows you to enable **geo-redundancy, multi-region writes, and availability zones** for the account. These options can also be configured post-deployment.

6. The Networking tab allows you to configure **network access and connectivity** for your Azure Cosmos DB account. There are three options to choose from for network configuration: *All networks*, *Public endpoint (selected network)*, and *Private endpoint*.
- All networks* opens access to the account to applications from any network. This option **removes network isolation** as a data security component to the Azure Cosmos DB configuration.
 - Public endpoint (selected network)* configures Azure Cosmos DB to use a firewall to only allow access from **certain IP addresses**. This includes access from the Azure Portal, the IP address of the machine that is creating the Azure Cosmos DB account, and IP addresses in one or more subnets in an Azure VNet. Figure 3.7 illustrates an example of this configuration.
 - Private endpoint* attaches an IP address in an Azure VNet to the Azure Cosmos DB account, limiting access to applications that can **communicate with the VNet**. This option also allows you to enable access to the account from the Azure Portal and the IP address of the machine that is creating the Azure Cosmos DB account.

FIGURE 3.7 Create an Azure Cosmos DB Account: Networking tab.

The screenshot displays the 'Networking' tab in the Azure Cosmos DB account configuration interface. At the top, there are tabs for 'Basics', 'Global Distribution', 'Networking' (selected), 'Backup Policy', 'Encryption', 'Tags', and 'Review + create'. Below the tabs, the 'Network connectivity' section explains that the account can be connected either publicly (via public IP addresses or service endpoints) or privately (using a private endpoint). Under 'Connectivity method', three radio buttons are shown: 'All networks' (unselected), 'Public endpoint (selected networks)' (selected), and 'Private endpoint' (unselected). The 'Configure Firewall' section has two rows: 'Allow access from Azure Portal' with 'Allow' (selected) and 'Deny' (unselected) radio buttons, and 'Allow access from my IP' with 'Allow' (unselected) and 'Deny' (selected) radio buttons. The 'Virtual Network' section includes a 'Virtual Network' dropdown showing '(new) cosmosnet' and a 'Subnet' dropdown showing '(new) cosmosdb ()'. A link 'Create a new virtual network' is also visible.

7. The Backup Policy tab allows you to select between a **Periodic or Continuous backup** strategy for data stored in this Azure Cosmos DB account. The *Periodic* setting allows you to set the **time interval, retention rate, and zone redundancy** for data backups. The *Continuous* setting will **automatically back up data within 100 seconds** of a change in the account, including those made to databases, containers, and items. Figure 3.8 illustrates an example of a *Periodic* backup policy configuration.

FIGURE 3.8 Create an Azure Cosmos DB Account: Backup Policy tab.


Backup policy ⓘ ☒ Periodic ☐ Continuous

Backup interval ⓘ Minute(s)
60-1440

Backup retention ⓘ Hours(s)
8-720

Copies of data retained 2

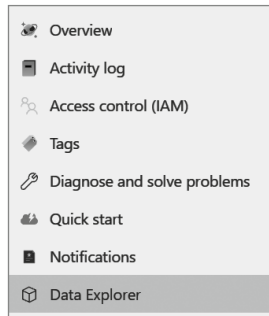
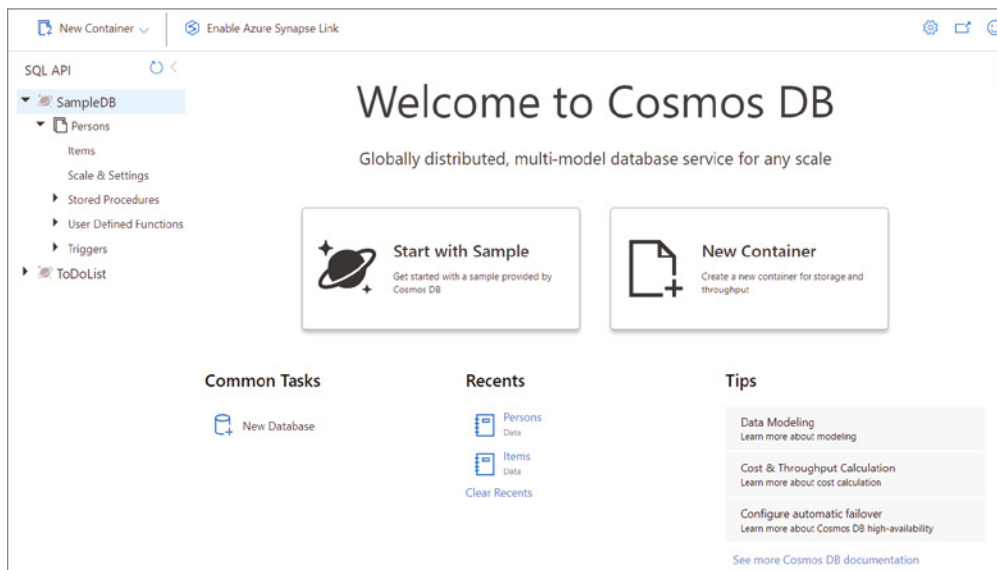
Backup storage redundancy ⓘ ☒ Geo-redundant backup storage
☐ Zone-redundant backup storage
☐ Locally-redundant backup storage

8. The Encryption tab allows you to choose whether **data encryption** uses a key that is generated and managed by Azure or a custom one that is stored in Azure Key Vault.
9. The **Tags** tab allows you to place a tag on the Azure Cosmos DB account for **cost management**.
10. Finally, the **Review + Create** tab allows you to **review** the configuration choices made during the design process. If you are satisfied with the choices made for the instance, click the Create button to begin provisioning the Azure Cosmos DB account.

Configuring Databases and Containers

After your Azure Cosmos DB account is deployed, you can start adding databases and containers to store data. You can add these objects using Azure PowerShell, Azure CLI, and Infrastructure as Code templates. There are also SDKs available in .NET, Java, Node.js, Python, and Xamarin that can be used to build Azure Cosmos DB objects. For the purposes of the DP-900 exam, let's focus on configuring these objects using the following steps in the Azure Portal:

1. Go to the **Azure Cosmos DB** page in the Azure Portal and click on the Azure Cosmos DB account you recently created.
2. In the left-side panel of the Azure Cosmos DB blade, **click Data Explorer**. Figure 3.9 shows where this button is located.
3. Data Explorer provides a **web-based interface** to **interact** with databases, containers, and items in Azure Cosmos DB. The splash page shows all the objects and data in that account, as well as some quick links to creating new tasks such as a new database. This page can be seen in Figure 3.10.
4. To create a new database, click New Database under **Common Tasks**. The New Database page gives you the option to **provision throughput** using the autoscale or manual options, as well as the required number of RU/s if using manual or the max database RU/s if using autoscale. Figure 3.11 is an example of a new database using the autoscale option with a maximum RU/s allocation of 4000.

FIGURE 3.9 Azure Cosmos DB Data Explorer button**FIGURE 3.10** Azure Cosmos DB Data Explorer splash page

5. To create a new container for that database, you can either click the New Container button in the upper-left corner of the Data Explorer page or you can click the ellipsis next to the database name and select *New Container*.
6. The New Container page gives you several options for configuring your container. Along with selecting the database that will host the container, you can give the container a name and set a partition key. This page also allows you to set whether the container will use dedicated or shared throughput. When selected, the *Provision dedicated throughput for this container* check box will provision dedicated throughput for the container. Otherwise, the container will share throughput with the other containers hosted by the database. Figure 3.12 shows the configuration for a container that uses dedicated throughput.

FIGURE 3.11 New Database

New Database

* Database id ⓘ

dp900cosmosdb01

☒ Provision throughput ⓘ

* Database throughput (autoscale) ⓘ

☒ Autoscale ☐ Manual

Estimate your required RU/s with [capacity calculator](#).

Database Max RU/s ⓘ

4000 *

Your database throughput will automatically scale from **400 RU/s (10% of max RU/s) - 4000 RU/s** based on usage.

Estimated monthly cost (USD) ⓘ: **\$35.04 - \$350.40** (1 region, 400 - 4000 RU/s, \$0.00012/RU)

OK

7. From here you can start **adding new items** to the container through Data Explorer or an application.



It is imperative that the partition key is chosen correctly from the start as it cannot be changed after the container is created. The only way to change the partition key is to create a new container with a new partition key and migrate the data from the old container to the new one.

FIGURE 3.12 New Container

New Container ✕

* Database id ⓘ
☐ Create new ☒ Use existing
 dp900cosmosdb01 ▼

* Container id ⓘ
 dp900cosmoscontainer01

* Partition key ⓘ
 /id

☒ Provision dedicated throughput for this container ⓘ

* Container throughput (400 - unlimited RU/s) ⓘ
☐ Autoscale ☒ Manual
 Estimate your required RU/s with [capacity calculator](#).
 400 *

Estimated cost (USD) ⓘ: **\$0.032 hourly / \$0.77 daily / \$23.36 monthly** (1 region, 400RU/s, \$0.00008/RU)

Unique keys ⓘ
 + Add unique key

Analytical store ⓘ
☐ On ☒ Off
 Azure Synapse Link is required for creating an analytical store container. Enable Synapse Link for this Cosmos DB account. [Learn more](#)
 Enable

> Advanced

OK

Configuring Global Distribution

Additional regions can be added to an Azure Cosmos DB account to replicate your data for high availability purposes. This can be done through the Azure Portal using the following steps:

1. Click the **Replicate Data Globally** button under Settings in the left-side panel of the Azure Cosmos DB blade.
2. From this page, you can **add regions** by either clicking on them in the **world map** or clicking **Add Region** and selecting one in the **drop-down list**. You can also choose to add a new write region using the Add Region button. Figure 3.13 illustrates how to add a new write region replica.

FIGURE 3.13 Adding a new write region replica

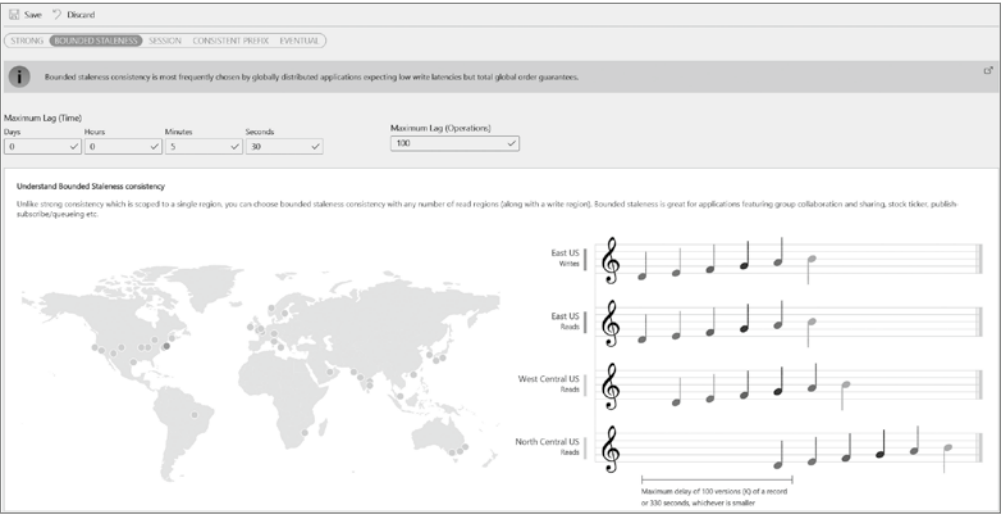


Configuring Consistency

The Azure Portal allows you to change the consistency level from the default session consistency to another. Simply click the Default Consistency button under Settings in the left-side panel and choose a new default consistency level.

If you choose the bounded staleness consistency level, you will be given the option to configure the **maximum lag time and maximum lag operations**. Figure 3.14 illustrates how you can change the default consistency to bounded staleness through the Default Consistency page.

FIGURE 3.14 Updating the default consistency to bounded staleness



Azure PowerShell

Just as with relational databases in Azure, you can use Azure PowerShell to create and manage all components of Azure Cosmos DB. The following PowerShell script can be run on the Azure Cloud Shell or a PowerShell window to create a new Azure Cosmos DB account:

```
<#
Sign into your Azure environment. Not required
if running this script in the Azure Cloud Shell
#>
Connect-AzAccount

<#
Set the parameters needed to create the account
such as the resource group name, account name,
API type, consistency level, and replica locations
#>
$resourceGroupName = "dp900cosmos001"
$accountName = "dp900cosmos001"
$apiKind = "Sql"
$consistencyLevel = "Session"
$locations = @()
$locations += New-AzCosmosDBLocationObject `
-LocationName "East US" -FailoverPriority 0 -IsZoneRedundant 0
$locations += New-AzCosmosDBLocationObject `
-LocationName "West US" -FailoverPriority 1 -IsZoneRedundant 0

#Create the account
New-AzCosmosDBAccount `
    -ResourceGroupName $resourceGroupName `
    -LocationObject $locations `
    -Name $accountName `
    -ApiKind $apiKind `
    -EnableAutomaticFailover: $true `
    -DefaultConsistencyLevel $consistencyLevel
```

This script includes a few key parameters that are used to define the account:

- **\$resourceGroupName**—The resource group that the Azure Cosmos DB account is going to be deployed to. The resource group must already exist.
- **\$accountName**—The name for the account.
- **\$apiKind**—The Azure Cosmos DB API that will be used for the account.
- **\$consistencyLevel**—The default consistency level for the account.

- **\$locations**—The replica regions for the account. The region with *FailoverPriority* set to 0 is the write region.

Azure PowerShell can also be used to create an Azure Cosmos DB database and container. The following script is used to create a new database in the newly created account with 4000 RU/s, as well as a container with 400 RU/s:

```
$resourceGroupName = "dp900cosmos001"
$accountName = "dp900cosmos001"
$databaseName = "dp900cosmosdb01"
$containerName = "dp900cosmoscontainer01"
$partitionKey = "/Id"
$databaseThroughput = 4000
$containerThroughput = 400
```

```
New-AzCosmosDBSqlDatabase `
    -ResourceGroupName $resourceGroupName `
    -AccountName $accountName `
    -Name $databaseName `
    -Throughput $databaseThroughput
```

```
New-AzCosmosDBSqlContainer `
    -ResourceGroupName $resourceGroupName `
    -AccountName $accountName `
    -DatabaseName $databaseName `
    -Name $containerName `
    -PartitionKeyKind Hash `
    -PartitionKeyPath $partitionKeyPath `
    -Throughput $containerThroughput
```

More information about creating and managing Azure Cosmos DB objects with Azure PowerShell can be found at <https://docs.microsoft.com/en-us/azure/cosmos-db/sql/manage-with-powershell>.

Azure CLI

Azure CLI is an **alternative to Azure PowerShell** for creating and managing Azure Cosmos DB components via a **scripting language**. The following Azure CLI script can be run on the Azure Cloud Shell, in a PowerShell window, or in a command prompt to create a new Azure Cosmos DB account, database, and container. This script uses the **same parameters** as the Azure PowerShell script in the previous section:

```
resourceGroupName='dp900cosmos001'
accountName='dp900cosmos001'
databaseName='dp900cosmosdb01'
```

```

containerName='dp900cosmoscontainer01'
partitionKey='/Id'
dbThroughput=4000
containerThroughput=400

```

```

az cosmosdb create \
  -n $accountName \
  -g $resourceGroupName \
  --default-consistency-level Session \
  --locations regionName='West US 2' \
    failoverPriority=0 isZoneRedundant=False \
  --locations regionName='East US 2' \
    failoverPriority=1 isZoneRedundant=False \

```

```

az cosmosdb sql database create \
  -a $accountName \
  -g $resourceGroupName \
  -n $databaseName \
  --throughput $dbThroughput

```

```

az cosmosdb sql container create \
  -a $accountName -g $resourceGroupName \
  -d $databaseName -n $containerName \
  -p $partitionKey --throughput $containerThroughput

```

More information about creating and managing Azure Cosmos DB objects with Azure CLI can be found at <https://docs.microsoft.com/en-us/azure/cosmos-db/sql/manage-with-cli>.

ARM Template

As mentioned in Chapter 2, ARM templates can be used to **define** the **resources and configuration requirements** for Azure deployments. These templates can be used to **automate new deployments** of Azure Cosmos DB as well as configuration changes for Azure Cosmos DB in different development environments.

One example of an ARM template that creates an Azure Cosmos DB account, database, and container can be found at <https://docs.microsoft.com/en-us/azure/cosmos-db/sql/quick-create-template?tabs=CLI>. The script can be deployed by clicking the Deploy To Azure button in the link and entering the required parameters or by running the following Azure PowerShell script:

```

Connect-AzAccount
$resourceGroupName = "dp900cosmos001"

```

```
$location = "East US"
```

```
New-AzResourceGroup
```

```
-Name $resourceGroupName
```

```
-Location $location
```

```
New-AzResourceGroupDeployment
```

```
-ResourceGroupName $resourceGroupName `
```

```
-TemplateUri https://raw.githubusercontent.com/Azure
```

```
/azure-quickstart-templates/master/quickstarts
```

```
/microsoft.documentdb/cosmosdb-sql/azuredeploy.json
```

Azure Cosmos DB Security

Data security for Azure Cosmos DB is implemented at multiple levels in Azure. Just as with data stored in a relational database in Azure, unauthorized access to Azure Cosmos DB is prevented using **network isolation** and **identity management**. Data stored in Azure Cosmos DB is also **encrypted at rest and in transit** to **protect data** from **malicious activity**. The following sections examine the methods Azure uses to secure data stored in Azure Cosmos DB in further detail.

Network Isolation

We briefly examined the two network isolation options for Azure Cosmos DB while going over how to deploy an account using the Azure Portal. These options include the following:

- Using the **Azure Cosmos DB firewall** to set IP-based access controls that **restrict communication** to an approved set of IP addresses. This can be taken a step further by allowing **access for entire subnets** by enabling the Azure Cosmos DB service endpoint on them.
- Assigning a **private IP address** from a VNet Azure Cosmos DB account with a private endpoint. This will restrict access to only applications that can communicate with the VNet that the private endpoint is associated with.

While there is an option to open Azure Cosmos DB access to requests from any network, it is important to consider the security implications, if any, of that setting. Rarely are security requirements satisfied with just access management and data encryption methods being put in place. Network isolation is an important design consideration and should be discussed when building a data-driven solution that uses Azure Cosmos DB.

Access Management

Azure Cosmos DB provides three approaches to control data access: **key-based** access control, **role-based** access control (RBAC), and **resource tokens**. Not only do each of these options **restrict access** to only users who should have access, they also determine whether the user has **read-write or read-only** access to database objects. The following sections provide an overview of these options.

Key-Based Access Control

Azure Cosmos DB provides a **primary** and a **secondary key** for **read-write access** as well as a primary and a secondary key for **read-only access**. Keys provide access to all resources in an Azure Cosmos DB account. The purpose of having a primary and a secondary key is to allow users to **regenerate** one key without requiring any **downtime**.

While keys can be useful when providing access to different applications, they can be **cumbersome** to manage. Keys also expose more Azure Cosmos DB account objects than what most users need. In most cases, it is better practice to use an identity management model that grants fine-grained permissions to **Azure Active Directory** (AAD) or **native Azure Cosmos DB identities** for database authentication and authorization.

Role-Based Access Control (RBAC)

Azure enables organizations to **centralize identity management** with **AAD and RBAC** roles. As discussed in Chapter 2, RBAC roles are used to control access to different Azure services. RBAC roles can be assigned to AAD objects (known as identities) such as users, groups, service principals, and managed identities, giving them the ability to perform tasks that are allowed by those roles. There are several RBAC roles specific to Azure Cosmos DB that can be used to perform management and data manipulation operations.

First, let's examine Azure Cosmos DB RBAC roles that control management plane operations. These roles allow AAD identities to manage create/replace/delete operations for Azure Cosmos DB account objects, database backups and restores, and performance monitoring. The following are the Azure Cosmos DB RBAC roles that support management operations:

- The *DocumentDB Account* **Contributor** role can manage Azure Cosmos DB accounts.
- The *CosmosDB Account* **Reader** role can read Azure Cosmos DB account data.
- The *Cosmos* **Backup Operator** role can submit a **restore request** for a periodic-backup-enabled database or container. It can **modify the backup interval and retention** through the Azure Portal. This role cannot access any data or use Data Explorer.
- The *Cosmos* **RestoreOperator** role can perform a **restore** for an Azure Cosmos DB account using the **continuous backup mode**.
- The *Cosmos* **DB Operator** role can provision Azure Cosmos DB accounts, databases, and containers. It cannot access any data or use Data Explorer.

More information about Azure Cosmos DB RBAC roles that support management activities can be found at <https://docs.microsoft.com/en-us/azure/cosmos-db/role-based-access-control>.

The next set of Azure Cosmos DB RBAC roles to bear in mind are those that support data plane operations. These allow AAD identities to create, read, update, and delete data from databases and containers. The following are the two built-in Azure Cosmos DB RBAC roles used to manage data plane operations:

- The *Cosmos DB Built-in Data* **Reader** role can read account metadata, data from specific items (point-reads and queries) and a specific container's change feed.
- The *Cosmos DB Built-in Data* **Contributor** role can read account metadata and perform create, read, and delete operations on data in specific containers and items.

More information about Azure Cosmos DB RBAC roles that support management activities can be found at <https://docs.microsoft.com/en-us/azure/cosmos-db/how-to-setup-rbac>.

Resource Tokens

Resource tokens allow **limited time access** to Azure Cosmos DB resources such as containers, partition keys, items, stored procedures, triggers, and user-defined functions. These tokens are initially created when a user is granted permissions to a specific resource and are valid for a **preset time limit**. The default time limit for a resource token is **one hour** and can be extended to a maximum of **five hours**. Resource tokens are re-created when a user makes an API request (GET, PUT, or POST) to Azure Cosmos DB.

Azure Cosmos DB database users are identity constructs that provide permissions to specific objects in a database, much like database contained users in Azure SQL. Users can be granted different levels of access to database resources using a set of permissions, also known as a permission resource. Permissions are authorization tokens associated with a database user that are used to authorize access to different database resources. Permission resources offer the following levels of access for database resources:

- *All*—This mode provides read, write, and delete access to a resource.
- *Read*—This mode provides read-only access to a resource.

Data Encryption

Data encryption **at rest and in transit** is provided out of the box for Azure Cosmos DB. There are no controls to turn encryption on or off. Azure Cosmos DB supports data encryption in transit with TLS version 1.2 or higher. Data stored in Azure Cosmos DB is encrypted at rest with keys that are managed behind the scenes by Microsoft. Organizations also have the option to add a second layer of encryption with their own keys.

Azure Cosmos DB Common Connectivity Issues

As with any data storage service, there will be times when issues occur when interacting with Azure Cosmos DB. These issues are typically related to bad request exceptions, unauthorized requests, or forbidden exceptions. The following sections include common Azure Cosmos DB connectivity issues and how to troubleshoot them.

Bad Request Exceptions

Errors that return the **HTTP 400 status code** represent bad request exceptions where the application request contains **invalid data** or is **missing required parameters**. These errors are typically caused by the following issues:

- The *missing the ID property* error means that the JSON item that is being inserted is missing the required ID property. **Specify the ID property** with a string value as a part of the item to resolve this issue.

- The *invalid partition key type* error means that the partition key value is an invalid data type. Make sure the partition key is a string or a numeric value.
- The *wrong partition key value* error means the partition key value being passed in the request does not match the item value in the container. For example, if the partition key path is /Id, the item has a field called Id and has values that do not include the value that was provided in the query. To resolve this issue, make sure the value used in the query is a valid partition key value.

Unauthorized Requests

Unauthorized requests are represented by a 401 HTTP error code, happening when an application request is performing an action with an invalid key. The following is a list of potential causes for unauthorized requests:

- The key is regenerated and the application using the key does not follow best practices for key rotation. This issue can be resolved by rotating the primary key to the secondary key and then regenerating the primary key. More information on key regeneration and rotation can be found at <https://docs.microsoft.com/en-us/azure/cosmos-db/secure-access-to-data?tabs=using-primary-key#key-rotation>.
- The key is misconfigured or was not copied correctly. Simply recopy the primary key or rotate to the secondary key to resolve this issue.
- The application is using a read-only key when trying to perform write operations. Switch the key to a read-write key if the application should be authorized to create or delete data.
- The application is trying to access a container before the container has finished being created. This typically happens when a container is deleted and then re-created with the same name.

Forbidden Exceptions

Forbidden exceptions occur when a data plane request comes from an application whose IP address is not whitelisted by the Azure Cosmos DB firewall or cannot communicate with the VNet the Azure Cosmos DB account is associated with. These exceptions are represented by 403 status codes.

Solutions to this error depend on if the application request comes from an IP address that can communicate with the Azure Cosmos DB account. They will also depend on what type of network isolation the account is using. Use the following recommendations if the application request is coming from an expected path:

- If the Azure Cosmos DB account is using the firewall, check to make sure the request's IP address is whitelisted in the Azure Cosmos firewall or is coming from a subnet with the Azure Cosmos DB service endpoint enabled.
- If the Azure Cosmos DB account is using a private endpoint, then make sure that the request's IP address can communicate with the VNet the private endpoint is associated with.

If the application request is not coming from an expected path, the issue is likely related to the application-side configuration. Use the following guidance to troubleshoot the issue depending on the type of network isolation the account is using:

- If an application request was expected to use a **service endpoint** but uses the **public Internet** instead, then check to see if the subnet the application's **IP address** is in has **enabled** the Azure Cosmos DB service endpoint.
- If an application request was expected to come through a private endpoint but instead comes from the public Internet, then **check** to see if the **DNS** the application is using can **resolve** the account endpoint to the private IP address associated with the **private endpoint**.

Management Tools

Azure offers two Azure Cosmos DB management tools that developers can use to **write and test queries** before adding them to applications. **Data Explorer** and the **Azure Cosmos DB Explorer** give developers and administrators the ability to **create** new resources and **manage** existing resources as well as optimize the cost-performance ratio for throughput. The following sections describe each of these tools in further detail.

Data Explorer

Data Explorer is a **development environment** available in the Azure Portal for **querying and managing Azure Cosmos DB**. It can be used to **create and delete resources** such as databases, containers, stored procedures, user-defined functions, and triggers. Developers can use query windows, like those in SSMS and Azure Data Studio, to write SQL statements that read, write, or delete data.



Developers using the Azure Cosmos DB Core (SQL) API can create Jupyter notebooks in Data Explorer to analyze and visualize data. Notebook commands can be written in Python or C#.

Administrators can also take advantage of Data Explorer to manage Azure Cosmos DB. Tasks such as **scaling throughput, modifying the indexing policy, and setting a Time to Live (TTL) period** can be handled using Data Explorer.

Azure Cosmos DB Explorer

Azure Cosmos DB Explorer is a **full screen extension** of the Data Explorer tool. It offers the same capabilities, such as creating new account objects, authoring queries, and scaling throughput. However, unlike Data Explorer, Azure Cosmos DB Explorer can be used **outside** of the Azure Portal. Users connecting to an Azure Cosmos DB account will only need one of the **read-write or read-only keys** that are generated with the account. This allows administrators to **restrict** who can modify data while still providing an **easy-to-use development environment** for developers to use.

Use the following steps to open the Azure Cosmos DB Explorer from the Azure Portal:

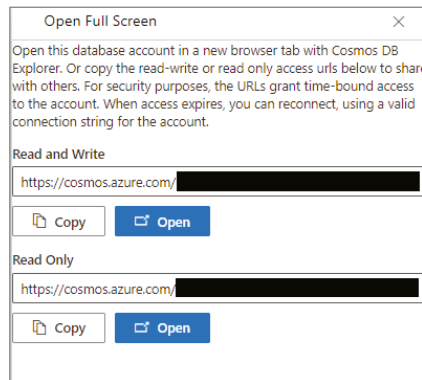
1. Go to the Azure Cosmos DB account created earlier and click on **Data Explorer**.
2. Click the **Open Full Screen icon** on the far right side of the Data Explorer blade. Figure 3.15 shows where you can find this icon.

FIGURE 3.15 Azure Cosmos DB Explorer Open Full Screen icon



3. After clicking this icon, you will be presented with two URL options: *Read and Write* and *Read Only*. You can either **copy** one of the links or click the **Open button** to open Azure Cosmos DB Explorer in a separate browser tab. Figure 3.16 illustrates what this pop-up page looks like.

FIGURE 3.16 Azure Cosmos DB Explorer pop-up screen



4. Once Azure Cosmos DB Explorer is opened, you will see the **same interface** as the Data Explorer.

Summary

This chapter covered NoSQL databases, including key-value, document, columnar, and graph databases. While they specialize in different scenarios, they do share some common characteristics. These include ambiguous implementations of ACID principles, flexible schema design, and the ability to scale horizontally.

Azure Cosmos DB is a fully managed, highly available PaaS NoSQL database that offers multiple database APIs for each type of NoSQL data store, including the Azure Cosmos DB

Table API, Azure Cosmos DB Core (SQL) API, Azure Cosmos DB API for MongoDB, Azure Cosmos DB Cassandra API, and Azure Cosmos DB Gremlin API. Each of these APIs comes with at least a 99.99 percent SLA and can easily be replicated to different regions all around the world. Azure Cosmos DB allows users to choose from five different consistency models to balance the trade-off between consistency, availability, and performance.

Compute is measured in Azure Cosmos DB as the throughput required to read and write data. Throughput is represented as Request Units per second (RU/s). RU/s can be set at the database and the container level by either provisioning a dedicated number of RU/s or setting a maximum number of RU/s that a database or container can scale to. There is also a serverless option that lets Azure Cosmos DB use as many RU/s as it needs for a workload.

Just like relational databases in Azure, Azure Cosmos DB can be deployed manually in the Azure Portal or automated with a script or an Infrastructure as Code template. Data stored in Cosmos DB is also secured at multiple layers, natively encrypting data at rest and in transit and offering flexible network isolation and access management options.

This chapter finishes by providing an overview of the two primary management tools that can be used to administer and query Azure Cosmos DB: Data Explorer and the Azure Cosmos DB Explorer. Data Explorer can be accessed in the Azure Portal. Azure Cosmos DB Explorer is a stand-alone web application that provides the same options and interface as Data Explorer and is typically used by developers who do not have access to the Azure Portal.

Exam Essentials

Describe the characteristics of NoSQL databases. NoSQL databases have become increasingly popular in the last several years due to larger volumes of data being produced at faster speeds. The advent of agile development standards and decreasing storage cost has also led to software developers being more empowered to use database platforms that offer more dynamic storage options.

The common characteristics between the different NoSQL database categories include schema flexibility, ambiguous interpretations of ACID principles, and the ability to scale horizontally.

Describe key-value stores. Key-value stores are the simplest type of NoSQL database. Each entry includes a data value and a unique key. These data stores are optimized for ingesting large volumes of data that must be stored and read very quickly. Azure provides several options to implement a key-value store, including Azure Table storage, Azure Cosmos DB Table API, and Azure Cache for Redis.

Describe document databases. Document databases are like key-value stores in that each entry includes a unique key with values of data. There are two options available for implementing a document database in Azure: Azure Cosmos DB Core (SQL) API and Azure Cosmos API for MongoDB.

Describe columnar databases. Columnar databases organize data in rows and columns like a relational database, but group columns into column families so that data remains denormalized. There are two options available for implementing a columnar database in Azure: Azure Cosmos DB Cassandra API and HBase in Azure HDInsight.

Describe graph databases. Graph databases are specialized databases that are used to store the relationships between different entities. Graph databases can be implemented in Azure using the Azure Cosmos DB Gremlin API.

Describe Azure Cosmos DB. Azure Cosmos DB is a PaaS NoSQL database in Azure that can be used to build key-value, document, columnar, and graph data stores. It is highly resilient, providing users with the ability to replicate data globally.

The highest level of management in Azure Cosmos DB is an account. One account can have one or more databases, which serve as the unit of management for a set of containers. Containers are the most fundamental unit of scalability in Azure Cosmos DB, storing database objects such as user-defined functions, stored procedures, and data. Data is referred to as items and is distributed into multiple partitions by running a hash algorithm on a selected partition key.

Describe Azure Cosmos DB consistency levels. Users can customize the trade-off between read consistency and performance by choosing one of five consistency levels for their Azure Cosmos DB accounts: strong, bounded staleness, session, consistent prefix, or eventual. Session is the default consistency level for Azure Cosmos DB and is suitable for most workloads.

Describe Azure Cosmos DB throughput. Request Units (RUs) represent the throughput required to read and write data in Azure Cosmos DB. Since RU usage is measured per second, throughput is set using the Request Units per second (RU/s) measurement.

Users can allocate a dedicated number of RU/s at the database level and at the container level or provide a maximum number of RU/s that Azure Cosmos DB can automatically scale to. They can also choose to forgo provisioning throughput and use serverless to enable Azure Cosmos DB to manage RU/s usage without user intervention.

Describe Azure Cosmos DB APIs. There are five APIs in Azure Cosmos DB that allow users to build key-value, document, columnar, and graph data stores in Azure. They include the Azure Cosmos DB API, Azure Cosmos DB Core (SQL) API, Azure Cosmos DB API for MongoDB, Azure Cosmos DB Cassandra API, and Azure Cosmos DB Gremlin API.

The Azure Cosmos DB Core (SQL) API is native to Azure Cosmos DB and is recommended for new applications that require high performance and global distribution and when migrating to Azure Cosmos DB from other database platforms.

Describe Azure Table storage. Azure Table storage is a key-value store in Azure that stores nonrelational, structured data. Tables can be created in an Azure storage account and can host one or more entities of data. Remember that the partition key and row key form the primary key for each entity in a table.

Describe deployment options for Azure Cosmos DB. It is important to understand how to deploy Azure Cosmos DB using the Azure Portal, Azure PowerShell, Azure CLI, and Azure PowerShell. Remember that you can use a free tier discount for one Azure Cosmos DB account per subscription where the first 1000 RU/s and 25 GB of storage are free.

The best way to manage deployments across multiple development environments is to script the deployment using an Infrastructure as Code template. This can be with an ARM, Terraform, or Bicep template.

Describe how to secure Azure Cosmos DB. Just like relational database offerings in Azure, Azure Cosmos DB has multiple layers of security. Network isolation is achieved by either whitelisting IP addresses or VNets in the Azure Cosmos DB firewall or attaching a private endpoint to the account. Remember that while Azure Cosmos DB provides keys, it is more often better to assign RBAC roles to AAD identities or use permission resources with native Azure Cosmos DB users for access management.

Describe management tools for Azure Cosmos DB. There are two management tools that can be used to manage account resources, develop queries, and handle administrative tasks: Data Explorer and Azure Cosmos DB Explorer. Data Explorer can be opened after clicking on the Azure Cosmos DB account in the Azure Portal. Azure Cosmos DB Explorer is a web-based tool that offers the same capabilities as Data Explorer.

Review Questions

1. Which of the following services in Azure can be used to build a key-value store?
 - A. Azure Table storage
 - B. Azure Cosmos DB
 - C. Azure Cache for Redis
 - D. All of the above
2. Is the italicized portion of the following statement true, or does it need to be replaced with one of the other fragments that appear below? “Queries can return data that is filtered by keys and data fields from *key-value stores*.”
 - A. Document databases
 - B. Graph databases
 - C. Azure Table storage
 - D. No change necessary
3. What resource is the fundamental unit of scalability for throughput and storage?
 - A. Database
 - B. Item
 - C. Container
 - D. Account
4. Which consistency level is the default consistency level for Azure Cosmos DB?
 - A. Session
 - B. Consistent prefix
 - C. Strong
 - D. Bounded staleness
5. You are designing a key-value store in Azure that will be used to store user sessions for an e-commerce site. The chosen data store must be able to allow writes to multiple regions to ensure low write latency for global users. Which Azure service is the best choice for this solution?
 - A. Azure Table storage
 - B. Azure Cosmos DB Graph API
 - C. Azure Cosmos DB Table API
 - D. Azure Key-Value Storage

6. Which of the following Azure Cosmos DB APIs allows users to host document databases?
 - A. Table API
 - B. Cassandra API
 - C. Gremlin API
 - D. API for MongoDB
7. Is the italicized portion of the following statement true, or does it need to be replaced with one of the other fragments that appear below? “The first *100 RU/s and 10 GB of storage* used by an Azure Cosmos DB account are free if you apply the free tier discount to it.”
 - A. 1000 RU/s and 25 GB of storage
 - B. 500 RU/s and 10 GB of storage
 - C. 2000 RU/s and 25 GB of storage
 - D. No change necessary
8. Which of the following options is the best way to manage Azure Cosmos DB deployments across multiple environments?
 - A. Azure PowerShell
 - B. Azure CLI
 - C. ARM templates
 - D. Azure Portal
9. What RBAC role can be used to restrict the data plane access of an Azure Active Directory identity to read-only?
 - A. Cosmos DB Built-in Data Contributor
 - B. CosmosDB Account Reader
 - C. Cosmos DB Built-in Data Reader
 - D. DocumentDB Built-in Data Reader
10. You are the administrator of an Azure Cosmos DB account that is used by an e-commerce application. Your manager has asked you to provide read-only access to one of the application developers and to recommend a tool that they can use to develop and test queries that they are adding to the application. The developer does not have access to the Azure Portal. Which of the following tools should you recommend?
 - A. Azure Cosmos DB Explorer
 - B. Data Explorer
 - C. Azure Storage Explorer
 - D. Visual Studio Code